Einsum Trees: An Abstraction for Optimizing the Execution of Tensor Expressions

Alexander Breuer
Friedrich Schiller University Jena
Jena, Germany
alex.breuer@uni-jena.de

Mark Blacher
Friedrich Schiller University Jena
Jena, Germany
mark.blacher@uni-jena.de

Max Engel
Friedrich Schiller University Jena
Jena, Germany
max.engel@uni-jena.de

Joachim Giesen
Friedrich Schiller University Jena
Jena, Germany
joachim.giesen@uni-jena.de

Alexander Heinecke
Intel Corporation
Santa Clara, CA, USA
alexander.heinecke@intel.com

Julien Klaus
Friedrich Schiller University Jena
Jena, Germany
julien.klaus@uni-jena.de

Stefan Remke
Friedrich Schiller University Jena
Jena, Germany
stefan.remke@uni-jena.de

Abstract

Einsum is a declarative language for tensor expressions that specifies an output tensor in terms of several input tensors. However, it does not specify how to compute the output tensor from the input tensors. A typical computational backend for the *einsum* language comprises two parts: First, a contraction path algorithm that breaks down an *einsum* expression into a sequence of binary tensor contractions. Second, the execution of the binary contractions. For efficient binary contractions, the data layout of the tensors must be optimized. So far, the computation of contraction paths and the optimization of the data layout for single, that is, *local*, binary tensor contractions have been studied in isolation. For optimizing the overall execution times of *einsum* expressions, we introduce *Einsum Tree IR*, an intermediate representation for *globally* optimizing the data layout for a given contraction path. We illustrate the effectiveness of the approach on a state-of-the-art Arm server processor, an x86 server processor, and an x86 desktop system.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Mathematics of computing** → **Mathematical software**.

Keywords: tensor compiler; einsum; intermediate representation; dense linear algebra

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716254

ACM Reference Format:
Alexander Breuer, Mark Blacher, Max Engel, Joachim Giesen, Alexander Heinecke, Julien Klaus, and Stefan Remke. 2025. Einsum Trees: An Abstraction for Optimizing the Execution of Tensor Expressions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3676641.3716254

1 Introduction

Tensor (hyper-)networks, which specify an output tensor in terms of a set of input tensors are ubiquitous in science and engineering. Applications range from the simulation of quantum circuits [7, 36, 41] to tensorized deep learning models [27, 40, 43, 52]. These applications can have tens to hundreds of thousands of input tensors.

Tensor hypernetworks can be succinctly defined by *einsum* expressions, which we introduce more formally in the background Section 2. Describing a tensor hypernetwork succinctly by an *einsum* expression, however, does not specify how to execute it efficiently on a given backend.

The evaluation of tensor hypernetworks with more than two input tensors is typically reduced to a sequence of binary tensor expressions, that is, expressions with only two input tensors. The binary expressions are scheduled in such a way that either the total number of floating point operations (flops) or the size of the largest intermediate tensor arising during the evaluation is minimized. A schedule for the binary expressions is also called a contraction path. Computing flop-optimal contraction paths is known to be an NP-hard problem [28]. However, finding good contraction path algorithms is an active area of research, which has combined various techniques to produce increasingly efficient contraction path algorithms [8, 18].

For any binary expression along a contraction path, the number of flops to compute the result tensor as well as the size of the result tensor, which is an intermediate tensor along the path, are fixed. The main goal for binary expressions is to minimize costly data movement. Optimizing binary tensor expressions for different computational backends is also an active area of research [12, 37, 49], where optimized data layouts facilitate efficient mappings to kernel routines.

So far, the global optimization problem of breaking an *einsum* expression flop-optimal or intermediate-size-optimal down into binary expressions, and the local optimization problems of minimizing costly data movements in binary tensor expressions have been studied in isolation. Data layout is not a concern of contraction path algorithms. Here, we introduce *Einsum Tree IR*, an intermediate representation for aligning the two optimizations, that is, the contraction path and the binary tensor expressions. *Einsum trees* are computed by first extracting a so-called contraction tree from a given contraction path. The tree is then transformed to optimize the data layout, without increasing the number of flops or the size of intermediate tensors. The resulting *einsum tree* can be traversed and its intermediate binary nodes, corresponding to binary expressions, efficiently mapped to nested loops over tensor processing primitives.

Specifically, this paper makes the following contributions:

- We propose *einsum trees* as an intermediate representation (IR) for optimizing the data layout for *einsum* expressions. *Einsum trees* build on a contraction path algorithm that minimizes either the number of flops or the size of the intermediate tensors. From the contraction path, a contraction tree is derived that abstracts away details of the path that do not affect either the number of flops or the sizes of the intermediate tensors. The contraction tree is then transformed into an *einsum tree* that encodes an optimized data layout for efficient execution of binary tensor expressions on different computational backends.
- We provide an implementation of *Einsum Tree IR* along with an optimization heuristic and a lowering to tensor processing primitives in the open source einsum_ir[1] software package.
- We evaluate[2] our *einsum tree* approach on three state-of-the-art testbeds: An Arm-based server processor, an x86 server CPU, and an x86 system targeting the desktop market. We achieve speedups of up to 2.1× over four state-of-the-art approaches on a set of real-world and complex contraction trees.

[1]The einsum_ir repository contains the source code and configurations for all the settings presented in this paper. It is available at https://doi.org/10.5281/zenodo.14843572.

[2]This paper is accompanied by a data repository containing scripts, raw results, post-processed results, and documentation of the experiments performed. It is available at https://doi.org/10.5281/zenodo.14843576.

2 Background

In this section, we first provide some background on the *einsum* notation and the contraction path problem for *einsum* expressions. As mentioned in the introduction, contraction paths are computed by minimizing a global objective, such as the number of flops to execute an *einsum* expression or the size of the largest intermediate tensor. Contraction paths break the execution down into binary contractions. Binary contractions can be executed by nested loops over tensor processing primitives. Therefore, we also provide background on tensor processing primitives (TPPs), which are used in our TPP-based binary tensor contraction backend.

2.1 Einsum

Since its introduction into *NumPy* in 2011 [19], *einsum* has established itself as a quasi-standard for specifying pure tensor expressions, that is, expressions without functions that operate elementwise on tensor entries. *Einsum* is now included in the deep learning frameworks PyTorch [42] and TensorFlow [1], NVIDIA's cuQuantum library for quantum information science [7], and various array programming libraries [45].

Einsum expressions are of the form

$$\texttt{einsum}\left(I_1, \ldots, I_n \rightarrow O;\ T_1, \ldots, T_n\right),$$

where the T_i are input tensors with multi-indices I_i, and O is the multi-index of the output tensor. The multi-indices are made up from tensor dimensions. The size of the multi-index I_i is the order of the tensor T_i. All dimensions that appear in multi-indices of at least two input tensors but not in the multi-index of the output tensor signify a contraction.

Table 1. *Einsum* expressions with two input matrices for classical linear algebra operations.

$\texttt{einsum}\left(pq, qr \rightarrow pr;\ A, B\right)$	matrix-matrix prod. $A \cdot B$
$\texttt{einsum}\left(pq, pq \rightarrow pq;\ A, B\right)$	Hadamard product $A \odot B$
$\texttt{einsum}\left(pq, pq \rightarrow;\ A, A\right)$	sq'd Frobenius norm $\|A\|_F^2$
$\texttt{einsum}\left(pq, qr \rightarrow pqr;\ A, B\right)$	Khatri-Rao product $A \circledcirc B$
$\texttt{einsum}\left(pq, rs \rightarrow pqrs;\ A, B\right)$	Kronecker product $A \otimes B$

The expressivity of *einsum* can be demonstrated with the examples in Table 1 from the domain of matrices, that is, tensors of order two. In the examples, only the matrix-matrix product and the squared Frobenius norm contain a contraction. The contraction is over the dimension q in the matrix-matrix product and over the dimensions p and q in the squared Frobenius norm, that is,

$$(A \cdot B)_{pr} = \sum_q A_{pq} B_{qr} \quad \text{and} \quad \|A\|_F^2 = \sum_p \sum_q A_{pq} A_{pq}.$$

In general, the multi-indices of tensors that appear in an *einsum* expression can be classified into the following four

types, whose naming convention generalizes the convention for BLAS-like routines:

R-dimension. The dimension appears in the multi-index of only one input tensor and not in the multi-index of the output tensor.

C-dimension. The dimension appears in the multi-indices of two or more input tensors and in the multi-index of the output tensor.

M/N-dimension. The dimension appears in the multi-index of only one input tensor and in the multi-index of the output tensor.

K-dimension. The dimension appears in the multi-indices of two or more input tensors, but not in the multi-index of the output tensor.

R-dimensions can be eliminated from an *einsum* expression by summing them out of the corresponding input tensor. For example, the dimension p in the expression

$$\texttt{einsum}\left(pq, qr \to r;\ A, B\right)$$

is an R-dimension. By the equivalence

$$\begin{aligned}&\texttt{einsum}\left(pq, qr \to r;\ A, B\right)\\ &\quad\equiv \texttt{einsum}\left(q, qr \to r;\ \texttt{einsum}\left(pq \to q;\ A\right), B\right),\end{aligned}$$

the dimension p can be summed out through the *einsum* expression $\texttt{einsum}\left(pq \to q;\ A\right)$.

C-dimensions amount to elementwise products, that is, generalizations of Hadamard products. The dimensions p and q in the Hadamard product in Table 1 are C-dimensions.

M/N-dimensions represent outer products. In Table 1, the dimensions p, q, r and s in the Kronecker product, and the dimensions p and r in the Khatri-Rao product are M/N-dimensions, whereas q is a C-dimension.

K-dimensions signify tensor contractions that generalize matrix-matrix products. In Table 1, the dimension q in the matrix-matrix product and both p and q in the squared Frobenius norm are K-dimensions.

2.2 Contraction Paths

The execution of an *einsum* expression is broken down into binary *einsum* expressions, that is, expressions with only two input tensors, by exploiting commutativity and a restricted form of associativity of *einsum* expressions [29]. For example, to break down tensor contractions of three or more tensors into binary contractions, intermediate tensors must be computed as in the following example

$$\begin{aligned}&\texttt{einsum}\left(ps, qs, rs \to pqr;\ A, B, C\right)\\ &\equiv \texttt{einsum}\left(pqs, rs \to pqr;\ \texttt{einsum}\left(ps, qs \to pqs;\ A, B\right), C\right)\\ &\equiv \texttt{einsum}\left(ps, qrs \to pqr;\ A, \texttt{einsum}\left(qs, rs \to qrs;\ B, C\right)\right)\\ &\equiv \texttt{einsum}\left(qs, prs \to pqr;\ B, \texttt{einsum}\left(ps, rs \to prs;\ A, C\right)\right).\end{aligned}$$

Note that the type of a dimension in a nested *einsum* expression can change. In the example, s is a K-dimension in all three outer expressions, but a C-dimension in the inner expressions. The example shows three possible reductions that give the same result but not at the same computational cost, in terms of the number of floating point operations (flops) or sizes of intermediate tensors. For example, if the sizes of the dimensions r and s are much larger than the sizes of the dimensions p and q, then the intermediate tensors

$$\texttt{einsum}\left(qs, rs \to qrs;\ B, C\right) \text{ and}$$
$$\texttt{einsum}\left(ps, rs \to prs;\ A, C\right)$$

are much larger than the intermediate tensor

$$\texttt{einsum}\left(ps, qs \to pqs;\ A, B\right),$$

and also significantly larger than the output tensor.

A *contraction path* schedules the binary contractions. Different contraction paths for the same expression can differ significantly in the number of flops needed to compute the contractions. Moreover, the size of the largest intermediate tensor can also be very different for different contraction paths. Computing optimal contraction paths is an NP-hard problem [28]. In practice, however, good contraction path algorithms are known [8, 18, 50].

A common data structure for storing contraction paths is a list of position tuples, known as *static single assignment* (SSA) format [5]. First, input tensors are assigned a position, typically by enumerating them. Then, once they are created, intermediate tensors are also assigned a position, typically by continuing the enumeration. For a path of the form $(0, 2), (1, 4), (1, 3), \ldots$, first the tensors at positions 0 and 2 are contracted, then the tensors at positions 1 and 4, and so on.

2.3 Tensor Processing Primitives

Once a contraction path, which breaks down an *einsum* expression into a sequence of binary *einsum* expressions, is given, it remains to execute the latter expressions. The execution of binary *einsum* expressions is typically reduced to matrix-matrix products. Our lowering to general matrix-matrix multiplications (GEMMs) builds on the assumption that *all tensors are stored contiguously using general row-major storage*. For example, suppose a three-dimensional tensor has dimensions p, q, and r of sizes 2, 3, and 4, respectively. Then the tensor holds a total of $2 \cdot 3 \cdot 4 = 24$ scalar values contiguously stored in memory. Here, we assume that if the tensor is indexed as pqr, then the dimension r has stride 1, q has stride 4, and dimension p has stride $3 \cdot 4 = 12$. That is, the order of the dimensions in the format string matters. Given a canonical matrix-matrix product

$$\texttt{einsum}\left(qp, rq \to rp;\ A, B\right),$$

when taking the order of the dimensions in the three index strings qp, rq and rp and the row-major storage format into account, then the dimension p has unit stride for the matrix A and is thus the *fast* dimension of A. The dimension q

is of type K, and the dimensions p and r are of type M/N. For matrix-matrix products, and more generally, for binary tensor expressions, we follow the BLAS convention [10] and refine the M/N-dimension type as

M-dimension. The dimension appears in the multi-index of the first (left) input tensor and in the multi-index of the output tensor.

N-dimension. The dimension appears in the multi-index of the second (right) input tensor and in the multi-index of the output tensor.

Algorithmically, a canonical matrix-matrix product is executed by the GEMM kernel

$$C[r][p] \mathrel{+}= A[q][p] * B[r][q],$$

which has the BLAS signature

$$C[N][M] \mathrel{+}= A[K][M] * B[N][K].$$

Assuming that the dimension sizes are given as $|p|$, $|q|$ and $|r|$, and being aware of our row-major storage, this means that we multiply the $|p| \times |q|$ matrix A with the $|q| \times |r|$ matrix B and update the $|p| \times |r|$ matrix C. Thus, when calling a GEMM routine in BLAS, we would use the parameters $M = |p|$, $N = |r|$, and $K = |q|$.

Our goal is to lower general binary *einsum* expressions

$$\texttt{einsum}\left(I_1, I_2 \rightarrow O;\ T_1, T_2\right),$$

to nested loops over only two types of general matrix-matrix multiplications (GEMMs), namely, **GEMM**s, with signature

$$C[N][M] \mathrel{+}= A[K][M] * B[N][K]$$

and **packed GEMM**s, with signature

$$C[N][M][C] \mathrel{+}= A[K][M][C] * B[N][K][C],$$

where the batch dimension of type C has unit stride.

Our goal can now be stated more concretely: lower a general binary *einsum* expression to nested loops over a (packed) GEMM such that at least a single M, N, and K dimension is covered by the (packed) GEMM. It turns out that this is not always possible. Therefore, we need the additional primitive **transpose**, which transposes the matrix.

In summary, we want to reduce general *einsum* expressions to a sequence of binary *einsum* expressions that can be executed by nested loops over three types of tensor processing primitives (TPPs), namely **GEMM**, **packed GEMM**, and **transpose**.

3 Lowering to Nested Loops over GEMMs

In the format string $I_1, I_2 \rightarrow O$ of a general binary *einsum* expression, the input tensor index string I_1 and the output tensor index string O can have additional M dimensions, the input tensor index string I_2 and the output tensor index string O can have additional N dimensions, the input tensor index string I_1 and the input tensor index string I_2 can have additional K dimensions, and all three strings can have, in contrast to the special case of a canonical matrix-matrix product, C dimensions. We assume that all R dimensions have already been summed out.

We need to decide which dimensions go into the (packed) GEMM, and which become loop dimensions, that is, dimensions that are used to loop over the (packed) GEMM. The following procedure labels dimensions either as GEMM dimensions or as loop dimensions:

1. Parse the dimensions in the output string O from right to left:
   While the dimension is of type C and coincides with the rightmost dimension in the input strings I_1 and I_2: label it as *GEMM type* and remove it from all three strings;
   While the dimension is of type M and coincides with the rightmost dimension in the input string I_1: label it as *GEMM type* and remove it from I_1 and from O;
   While the dimension is not of type N: label it as *loop* type and remove it from O.
2. Parse the dimensions in the input string I_1 from right to left:
   While the dimension does not coincide with the rightmost dimension of I_2: label it as *loop* type and remove it from I_1.
3. Parse the dimensions in the input string I_2 from right to left:
   While the dimension is of type K and coincides with the rightmost dimension in the input string I_1: label it as *GEMM type* and remove it from I_1 and from I_2;
   While the dimension does not coincide with the rightmost dimension of O: label it as *loop* type and remove it from I_2.
4. Parse the dimensions in the output string O from right to left:
   While the dimension is of type N and coincides with the rightmost dimension in the input string I_2: label it as *GEMM type* and remove it from I_2 and from O.
5. Label all remaining dimensions as *loop* type.

In the format string, let g_M, g_N, g_K, and g_C, be the contiguous substrings of GEMM dimensions of type M, N, K, and C, respectively. If a format string matches the mask

$$[...]g_K[...]g_Mg_C, [...]g_N[...]g_Kg_C \rightarrow [...]g_N[...]g_Mg_C,$$

then it can be mapped to nested loops over a (packed) GEMM.

We demonstrate this with four examples. The first example is a batched matrix-matrix multiplication, given by the format string $pqr, psq \rightarrow psr$. Here, we need to determine the dimension type of the indices p, q, r and s and to label them either as GEMM or as loop dimension. According to the definition, the dimension types are C for p, K for q, M for r, and N for s. The labels, GEMM or loop dimension, are assigned by the labeling algorithm as shown in Table 2. By the labeling algorithm, the dimension r is a GEMM dimension

Table 2. Walkthrough of the labeling algorithm on the example format string $pqr, psq \rightarrow psr$. The algorithm parses the output string $O =$ "psr" and the input strings $I_1 =$ "pqr" and $I_2 =$ "psq" from right to left. The current index is highlighted.

Step	Format String	Labeling
1.	$pqr, psq \rightarrow ps\mathbf{r}$ $pqr, psq \rightarrow ps\mathbf{r}$ $pq, psq \rightarrow p\mathbf{s}$	r: GEMM type
2.	$p\mathbf{q}, psq \rightarrow ps$	
3.	$pq, ps\mathbf{q} \rightarrow ps$ $p, p\mathbf{s} \rightarrow ps$	q: GEMM type
4.	$p, ps \rightarrow p\mathbf{s}$	s: GEMM type
5.	$p, p \rightarrow p$	p: loop type

of type M, q is a GEMM dimension of type K, s is a GEMM dimension of type N, and p is a loop dimension of type C. The resulting GEMM multiplies an $|r| \times |q|$ matrix with a $|q| \times |s|$ matrix to update an $|r| \times |s|$ matrix. If we denote the first matrix by A, the second matrix by B, and the result matrix by C, then the GEMM primitive becomes

$$C[s][r] \mathrel{+}= A[q][r] * B[s][q].$$

The second example is a packed matrix-matrix multiplication, given by the format string $pqr, spr \rightarrow sqr$. Here, q is of type M, s is of type N, p is of type K, r is of type C, and all are GEMM dimensions. The resulting tensor processing primitive is a packed GEMM primitive

$$C[s][q][r] \mathrel{+}= A[p][q][r] * B[s][p][r].$$

The third example is given by the format string $pqrs, ptq \rightarrow ptrs$. Here, the contiguous substring rs contains the GEMM dimensions r and s of type M, t is a GEMM dimension of type N, q is a GEMM dimension of type K, and p is a loop dimension of type C. The resulting GEMM again multiplies two matrices A and B into a result matrix C, but *fuses* the GEMM dimensions r and s. The GEMM primitive thus becomes

$$C[t][r \cdot s] \mathrel{+}= A[q][r \cdot s] * B[t][q].$$

However, there are also examples where the lowering fails, such as the expression given by the format string $pqr, psq \rightarrow prs$. Here, s is a GEMM dimension of type N, q is a GEMM dimension of type K, and p and r are loop dimensions of type C and M, respectively. The resulting GEMM primitive

$$C[s][1] \mathrel{+}= A[q][1] * B[s][q]$$

computes a matrix-vector product.

The fifth example is given by the format string $rqp, qsp \rightarrow prs$. According to our labeling scheme, the dimension s is a GEMM dimension of type N, while the other dimensions p, q and r are all loop dimensions of types C, K, and M, respectively. The mapping failed and the primitive becomes

$$C[s][1] \mathrel{+}= A[1][1] * B[s][1],$$

which is a degenerate GEMM that scales a vector by a scalar.

In this paper we say that the mapping *fails* if one of the substrings g_M, g_N or g_K is empty but loop dimensions of the corresponding type exist. Format strings for which the mapping to nested loops over (packed) GEMMs fails must be transformed using permutations to make the mapping work.

For the example $rqp, qsp \rightarrow prs$, one option is to internally compute the expression with the format string $pqr, psq \rightarrow psr$, which is our first example. This requires three permutations, the permutation of the left input index string I_1: $rqp \rightarrow pqr$, the permutation of the right input index string I_2: $qsp \rightarrow psq$, and the permutation of the output index string O: $psr \rightarrow prs$. In summary, we would evaluate the following expression: $((rqp \rightarrow pqr), (qsp \rightarrow psq) \rightarrow psr) \rightarrow prs$. Conceptually, this is related to an approach known as Transpose-Transpose-GEMM-Transpose (TTGT) [49].

We have discussed the execution of a single binary *einsum* expression by nested loops over a TPP. However, we are dealing with general *einsum* expressions decomposed into a sequence of binary *einsum* expressions along a contraction path. Our task is not to optimize the execution of a single binary *einsum* expression, but a whole sequence of related binary *einsum* expressions. In the following, we describe our approach to optimizing the execution of the sequence of binary *einsum* expressions from the contraction path of a general *einsum* expression. The optimization builds on an intermediate representation, described in the next section.

4 Intermediate Representation

The intermediate representation (IR) has a global view on a sequence of binary *einsum* expressions. The accompanying optimizations formulate each binary *einsum* expression in such a way so that they can be efficiently handled by the TPP backend. From a compiler perspective, the IR falls into the category of directed acyclic graph-based IRs [32]. Tensor operations are represented by IR nodes with one or two children. Nodes with one child permute the dimensions of a tensor, while nodes with two children represent binary *einsum* expressions. The leaf nodes are the input tensors, and the result at the root node is the output tensor.

4.1 Contraction Trees

From any contraction path, a partial order can be defined on the input and intermediate tensors where, one tensor precedes another if the first tensor is needed to compute the second one. The partial order can be encoded in a rooted, binary tree called a *contraction tree*. The leaves of a contraction tree are labeled by the index strings of input tensors, its inner nodes are labeled by index strings of intermediate tensors, and its root by the index string of the output tensor. A contraction tree does not uniquely determine a contraction path,

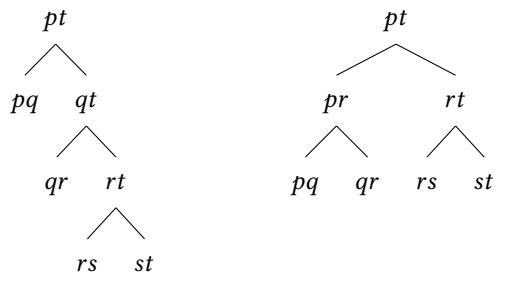


Figure 1. Contraction trees for the matrix chain multiplication problem $A \cdot B \cdot C \cdot D$. The tree on the left corresponds to the binary contractions $A_{pq} \cdot (B_{qr} \cdot (C_{rs} \cdot D_{st}))$, and the tree on the right corresponds to $(A_{pq} \cdot B_{qr}) \cdot (C_{rs} \cdot D_{st})$.

but all contraction paths leading to the same contraction tree have the same number of flops. For example, consider the matrix chain product $A \cdot B \cdot C \cdot D$, which can be expressed in *einsum* notation as

$$\texttt{einsum}\left(pq, qr, rs, st \rightarrow pt;\ A, B, C, D\right).$$

The matrix chain product can be broken down by a contraction path algorithm into binary contractions, for example, as $A \cdot (B \cdot (C \cdot D))$ or as $(A \cdot B) \cdot (C \cdot D)$. The corresponding *contraction trees* are shown in Fig. 1. In the first case, the tree determines the contraction order, whereas, in the second case, the tree does not determine the contraction order of the dimensions q and s, that is, it does not determine whether $A \cdot B$ is computed before $C \cdot D$, or vice versa. However, the number of flops and the size of the intermediate matrices do not depend on the contraction order between $A \cdot B$ and $C \cdot D$. Therefore, contraction trees abstract away details from the contraction path that do not affect the number of flops and the size of intermediate tensors.

4.2 Einsum Tree IR

The Einsum Tree IR distinguishes between equivalent contraction trees by two additional properties. First, swapping the children at binary nodes in a contraction tree, corresponding to binary contractions, results in an equivalent contraction tree. Our IR, however, distinguishes between these trees.

Second, in contrast to contraction trees, the order of the dimensions in the index strings at the nodes matters, that is, permuting that order results in a new IR tree. The order matters, because we assume that all tensors are stored contiguously using general row-major storage.

Furthermore, the dimensions in the index strings at the binary nodes of the tree are labeled by their dimension type, that is, C, M, N, or K, and by using the classification procedure of Sec. 3 as either GEMM or loop dimensions.

The goal is to apply transformations to a tree formulated in the IR such that the binary *einsum* expressions at the binary nodes of the tree can be mapped to nested loops over a (packed) GEMM such that at least a single M, N and K dimension is covered by the (packed) GEMM.

5 Optimizing Einsum Trees

The mapping of binary nodes to nested loops over (packed) GEMMs may fail. Therefore, a tree in the IR can be transformed using the following transformations:

1. **Swap** the left and right children of a binary node.
2. **Reorder** the index string of an interior node.
3. **Insert** a permutation node, that is, a node with a single child and a single parent.

For global optimization, we traverse the IR tree and transform the IR nodes such that the corresponding binary *einsum* expressions map to nested loops over either a GEMM or a packed GEMM.

For a binary node of the IR tree, let O be the index string at the node, I_1 be the index string at the left child node, and I_2 be the index string at the right child node. For optimizing the IR tree, we reorder the index strings at the child nodes. Since O is the output index string of the binary *einsum* expression

$$\texttt{einsum}\left(I_1, I_2 \rightarrow O;\ T_1, T_2\right),$$

which is represented by the node, it does not contain any K-dimension. We distinguish two cases, depending on the type of the rightmost dimension in O. Let d be the rightmost dimension in O.

1. d is of type C.

   Parse O from the right up to the first non-C-dimension and collect the dimensions in a contiguous index substring g_C. *[g_C is of GEMM-type].*

   If the first non-C-type dimension is of type N, then **swap** the children of the node. *[The type of the first non-C-type dimension is switched from N to M.]*

   Continue parsing from the right up to the first non-M-type dimension and collect the dimensions in a contiguous index substring g_M. *[g_M is of GEMM-type].*

   If the first non-M-type dimension is *not* of type N, then continue parsing from the right up to the first N-type dimension. *[The skipped dimensions are loop dimensions.]*

   Continue parsing from the right up to the first non-N-type dimension or the end of the index string and collect the dimensions in a contiguous index substring g_N. *[g_N is of GEMM-type].*

   Reorder the index string I_1 such that its rightmost substring is $g_K g_M g_C$, where g_K is a contiguous substring of all type-K dimensions.

   Reorder the index string I_2 such that its rightmost substring is $g_N g_K g_C$.

Now, after reordering, the binary *einsum* expression maps to nested loops over a **packed GEMM**.

2. d is of type N or of type M.

   If d is of type N, then **swap** the children of the node. *[The type of d is switched from N to M.]*

   Parse O from the right up to the first non-M-type dimension and collect the dimensions in a contiguous index substring g_M. *[g_M is of GEMM-type].*
   If the first non-M-type dimension is *not* of type N, then continue parsing from the right up to the first N-type dimension. *[The skipped dimensions are loop dimensions.]*

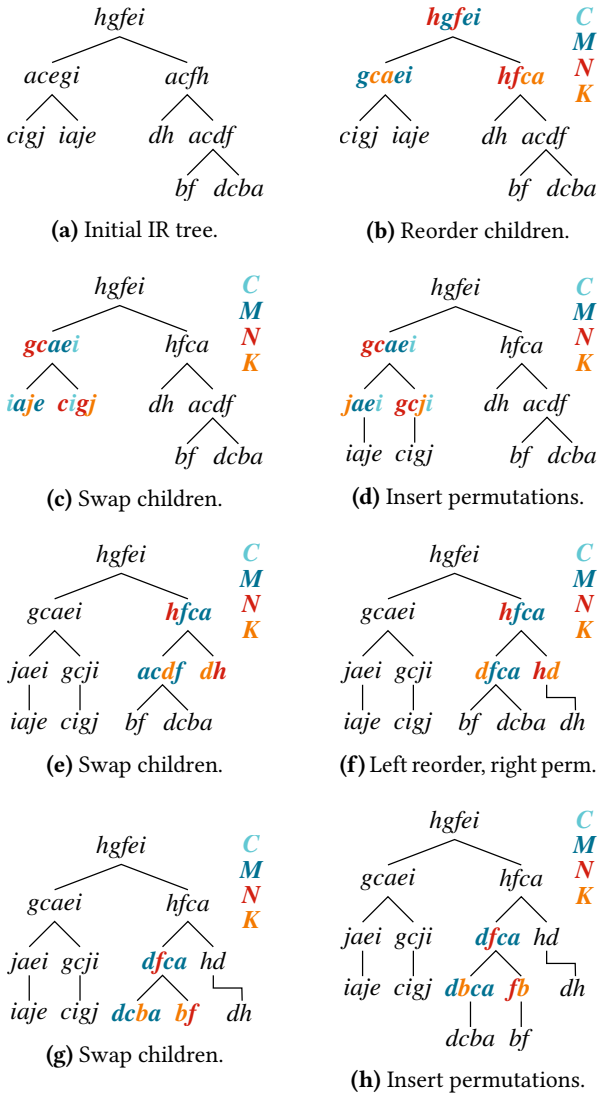
   Continue parsing from the right up to the first non-N-type dimension or the end of the index string and collect the dimensions in a contiguous index substring g_N. *[g_N is of GEMM-type].*

   Reorder the index string I_1 such that its rightmost substring is $g_K g_M$, where g_K is a contiguous substring of all type-K dimensions.

   Reorder the index string I_2 such that its rightmost substring is $g_N g_K$.

Now, after the reordering, the binary *einsum* expression maps to nested loops over a **GEMM**.

We traverse the IR tree in preorder starting at the root and transform the nodes as described above. At the leaf nodes of the transformed IR tree we realize the reordering of the index strings by **insert**ing permutation nodes. The permutation nodes themselves can be implemented by our third tensor processing primitive **transpose**. The whole optimization procedure is demonstrated exemplarily in Figure 2.



(a) Initial IR tree. **(b)** Reorder children. **(c)** Swap children. **(d)** Insert permutations. **(e)** Swap children. **(f)** Left reorder, right perm. **(g)** Swap children. **(h)** Insert permutations.

Figure 2. Illustration of the IR tree optimization procedure. Starting at the root, the procedure optimizes all tensor contractions by using the three simple transformations **swap**, **reorder**, and **insert** permutation nodes.

6 Performance Evaluation

In this section, we analyze the performance of the proposed einsum tree optimization heuristic and lowering to nested loops over tensor processing primitives. We answer three main questions:

- *Question:* What are the advantages of the proposed tree optimization heuristic (Sec. 5) when applied to a single contraction node of an einsum tree? *Answer:* In Sec. 6.3, we observe that despite its simplicity, the contraction performance of the approach is highly competitive and has short compile times.
- *Question:* Can we materialize these single-contraction speedups when executing complex contraction trees? *Answer:* Yes, in Sec. 6.4 we observe that our Einsum Tree IR optimization together with the TPP backend outperforms four state-of-the-art approaches for most of the tested real-world examples (Tab. 5). Speedups of up to 2.1× are possible.
- *Question:* Do we achieve performance portability by relying on tensor processing primitives in our backend? *Answer:* Yes, we observe high performance for different instruction set extensions (AVX512, Neon) and the three hardware testbeds introduced in Sec. 6.1.

6.1 Hardware and Software

We use three current and diverse hardware testbeds for performance evaluation. The testbeds cover the server and desktop segments and have different instruction set extensions for accelerating floating-point workloads:

Grace NVIDIA Grace CPU with 72 Arm Neoverse V2 cores supporting Arm Neon and 128-bit SVE2 vector instructions. Each core has 64 KiB of L1-instruction cache, 64 KiB of L1-data cache, and 1 MiB of L2 cache. The L3 cache is 114 MiB in size. The dual socket testbed is equipped with a total of 480 GiB of LPDDR5X-4800 memory. During all tests, the cores of the second socket were set offline by the operating system.

SPR Intel Xeon Platinum 8488C with 48 cores supporting AVX512 vector instructions. Each core has 32 KiB of L1-instruction cache, 48 KiB of L1-data cache, and 2 MiB of L2 cache. The L3 cache is 105 MiB in size. The testbed is equipped with a total of 192 GiB of DDR5-4800 memory.

Ryzen AMD Ryzen 7 8700G SoC with eight Zen 4 cores supporting AVX2 and AVX512 vector instructions. Each core has 32 KiB of L1-instruction cache, 32 KiB of L1-data cache, and 1 MiB of L2 cache. The L3 cache has a size of 16 MiB. The used testbed is equipped with 64 GiB of DDR5-5200 memory.

We have implemented the open-source C++ software einsum_ir for optimizing and lowering the proposed Einsum Tree IR following the ideas described in Sec. 3, Sec. 4, and Sec. 5. The software uses tensor processing primitives (TPPs) implemented in the LIBXSMM library [22] as its primary backend and OpenMP to parallelize over loop dimensions of type C, M, and N. We use the abbreviation **ET-TPP** to refer to einsum_ir with the TPP backend.

In addition, users can choose TBLIS [37] as an alternative backend for the binary nodes (**ET-TBLIS**). In this case, the entire binary tensor contraction is computed by TBLIS, including shared memory parallelization. However, we still use our compilation approach to optimize the data layouts of intermediate tensors in an einsum tree. For the presented results, we used the git hash 2cbdd21 of the TBLIS repository.

We also compare einsum_ir's performance with that of three established software packages. The abbreviation **TVM-Ansor** refers to the TVM AutoScheduler (0.18.0), also known as Ansor [60]. Specifically, we exported the compiled einsum trees from our IR to TVM's tensor expression language. We used 1000 measurement trials for the autotuning process, as recommended in the TVM documentation, and the default timeout of 10 s.

The second software package is **ATen**, PyTorch's tensor library (2.5.1). The ATen runs were performed by calling the at::einsum function and passing it a contraction path representing the contraction tree.

opt_einsum (3.4.0) [8] is the last software package we used. In this case, we used the function contract_expression to generate an expression with input data reflecting the respective contraction tree. opt_einsum supports several backends that can be used to perform binary tensor contractions, as long as they support tensordot and transpose operations. We used PyTorch (2.5.1) as the backend for the results presented here and use the abbreviation **OE-Torch** for the combination of the two.

For all runs, we first invoked the respective contraction tree execution once as a warm-up and measured the actual floating-point throughput in a separate run. The rationale behind this step is to flush the caches and ensure that any just-in-time code generation in downstream libraries without explicit compilation steps, such as BLAS libraries, is done first. Note that we already generate the code for ET-TPP when lowering to nested loops over GEMMs, i.e. it is part of a separate compilation step. We have also pinned the OpenMP threads to physical cores for the ET-TPP, ET-TBLIS, ATen, and OE-Torch approaches. For ET-TPP, we spawned 96 threads on SPR, or two threads per core, as this improved performance in this case. Elsewhere, there was no improvement from overloading the cores. TVM-Ansor has its own scheduling policies, which we left at their default settings.

Table 3. Measured FP32 performance on the three testbeds for a large SGEMM call and a corresponding blocked tensor contraction in ET-TPP.

Testbed	Approach	Dimension Sizes	GFLOPS
Grace	NVPL	2048,2048,2048	6502
	ET-TPP	8,256,128,16,32,64	5978
SPR	oneDNN	2048,2048,2048	6675
	ET-TPP	8,256,128,16,64,32	6370
Ryzen	OpenBLAS	2048,2048,2048	1011
	ET-TPP	8,256,64,32,16,128	1121

6.2 Achievable Performance

The first set of tests determines the maximum achievable floating-point performance of the three hardware testbeds. Our first test performs a large matrix-matrix multiplication by calling the BLAS routine SGEMM: $ab, ca \rightarrow cb$ with dimension sizes $|a| = 2048$, $|b| = 2048$ and $|c| = 2048$. Additionally, we assume that the matrix product is part of a multilayer perceptron and that we can choose a blocked data layout for all three matrices [14]. So we block the dimensions $a_1a_2b_1b_2, c_1c_2a_1a_2 \rightarrow c_1c_2b_1b_2$ with $|a_1|\cdot|a_2| = |a|$, $|b_1|\cdot|b_2| = |b|$ and $|c_1|\cdot|c_2| = |c|$ and assume an optimized data layout for the weights and activations: $b_1a_1a_2b_2, c_1a_1c_2a_2 \rightarrow c_1b_1c_2b_2$. The second test runs this contraction using the lowering to nested loops over small GEMMs as introduced in Sec. 3.

We used NVIDIA Performance Libraries BLAS (0.2.0) on Grace, oneDNN (3.6.1) on SPR, and OpenBLAS [58] version 0.3.28 on the Ryzen system. For the ET-TPP approach, we used a grid search on the block sizes given by the dimension sizes $|a_2|$, $|b_2|$ and $|c_2|$, and report the best obtained configuration for each system. The grid search tried all combinations of the three block sizes with possible values for b_2 and c_2 in $[16, 32, 64, 128]$, and possible values for a_2 in $[16, 32, 64, 128, 256]$. Tab. 3 shows the measured performance for the two settings on the three systems. The column "Dimension Sizes" provides the size of the dimensions in the binary tensor contractions in alphabetical order. This means that for SGEMM, the order is $|a|, |b|, |c|$, and for the blocked tensor contraction, the order is $|a_1|, |a_2|, |b_1|, |b_2|, |c_1|, |c_2|$. We ran all tests 100 times and report the mean performance.

We see that ET-TPP is competitive with the optimized BLAS libraries and even outperforms OpenBLAS on the Ryzen system. The best-performing block sizes are $a_2 = 256$, $b_2 = 16$ and $c_2 = 64$ for Grace, $a_2 = 256$, $b_2 = 16$ and $c_2 = 32$ for SPR, and $a_2 = 256$, $b_2 = 32$ and $c_2 = 128$ for Ryzen.

To achieve consistent performance with these results, we use a related strategy to deliberately limit the number of GEMM dimensions when lowering to nested loops over a (packed) GEMM (see Sec. 3). We fuse dimensions of the same type until a lower bound is reached, and stop before an upper bound is exceeded. All rejected dimensions become loop-type dimensions. Specifically, we use a lower bound of 4 and an upper bound of 16 for C-type dimensions, a lower bound of 32 and an upper bound of 512 for K-type dimensions, a lower bound of 32 and an upper bound of 128 for M-type dimensions, and a lower bound of 12 and an upper bound of 64 for N-type dimensions. These heuristic values were determined manually by benchmarking the performance of the corresponding GEMM kernels on a single core.

6.3 Single Binary Node

The second set of tests examines the performance of a single binary node in Einsum Tree IR. Specifically, we assume that the binary node has two intermediate tensors as children, which means that we can simply reorder the dimensions of the children at compile time. In Fig. 2a, the binary node acegi,acfh→hgfei represents such a case where we reorder the children to obtain the IR tree in Fig. 2b.

We use a modified version of the TCCG settings for the performance evaluation in this section. The TCCG benchmark problems are given by a set of binary contractions introduced by Springer and Bientinesi [49]. The number of dimensions in the tensors ranges from two to five, and the dimension sizes range from 20 to 7,248. We have made two changes to the original settings: First, we split large dimensions with sizes greater than 100 into two dimensions. This allows us to use small GEMMs as primitives and enables blocking of the input tensors, similar to the multilayer perceptron example in Sec. 6.2. Second, we followed the optimization heuristic presented in Sec. 5 together with the limits introduced in Sec. 6.2 for reordering the input tensors of the settings. Note that the data layout of the output tensor is preserved. Tab. 4 gives the specifications of the 24 studied contractions.

The Figures 3, 4 and 5 show the obtained performance on the Grace, SPR, and Ryzen testbeds. For ET-TPP, ET-TBLIS, ATen, and OE-Torch, we repeatedly ran each setting 100 times and report the mean performance. For TVM-Ansor, we followed the Auto-scheduling documentation and report the median of the built-in time evaluator with a minimum repetition time of 60 s.

We observe a general trend of increasing floating-point throughput across the tested software and hardware with increasing TCCG IDs. This is because we retained the original order of the TCCG settings, which Springer et al. empirically sorted by measured floating-point throughput [49]. The resulting order largely reflects the arithmetic intensity of the settings, as shown in Tab. 4.

Studying the Grace performance in detail (Fig. 3), we see that TVM-Ansor outperforms the other approaches for settings 1-5 and 8-12 but fails to find very good solutions for

Table 4. The table comprises the benchmarked single-node contractions. The column "ID" refers to the corresponding TCCG setting in Figures 3, 4 and 5. The dimension sizes are listed in alphabetical order. The "Ari. Int." column gives the arithmetic intensity in FP32 operations per byte.

ID	Contraction	Dimension Sizes	Ari. Int.
1	abdfe,cf→abcde	48,36,24,36,48,36	7
2	acdfe,bf→abcde	48,24,36,36,48,36	7
3	abed,ce→abcd	96,84,24,96,96	10
4	abcfe,df→abcde	48,36,36,24,48,48	8
5	aced,be→abcd	96,24,84,96,84	9
6	jki,efghjk→efghi	6,64,6,64,24,6,64	11
7	abed,ce→abcd	96,84,24,84,96	10
8	bcgf,adeg→abcdef	24,20,20,24,20,20,24	12
9	bdgf,aceg→abcdef	24,20,20,24,20,20,24	12
10	acgf,bdeg→abcdef	24,20,20,24,20,20,24	12
11	abgf,cdeg→abcdef	24,20,20,24,20,20,24	12
12	efhjki,gjk→efghi	6,64,24,4,94,6,64	11
13	bced,ae→abcd	96,84,84,84,96	24
14	aced,be→abcd	96,84,84,84,96	22
15	abed,ce→abcd	96,84,84,84,96	22
16	aedc,ebd→abc	96,84,84,84,96	41
17	gkiljh,ekilfj→efgh	6,64,4,94,6,64,6,64	95
18	gikljh,eiklfj→efgh	6,64,4,94,4,94,6,64	95
19	efiklj,ghkl→efghij	6,64,4,94,4,94,6,64	95
20	efiklj,ghkl→efghij	6,64,6,64,4,94,4,94	95
21	fihg,dieh→defg	151,48,181,40,151,48	1207
22	cefd,aebf→abcd	96,84,84,84,84,96	1283
23	aefd,becf→abcd	96,84,84,84,96,96	1336
24	cfed,afbe→abcd	96,84,84,96,84,8	1283

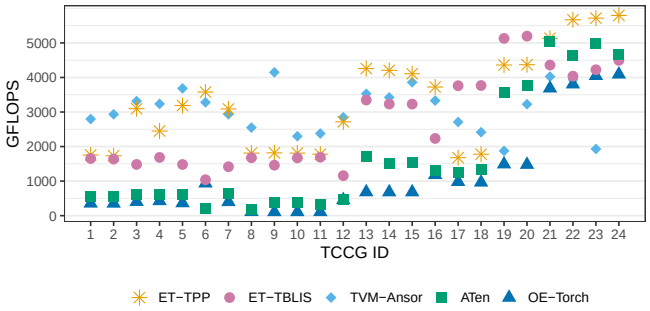


Figure 3. Performance of the five studied approaches for the modified TCCG settings on the Grace testbed. Performance is expressed in FP32 GFLOPS.

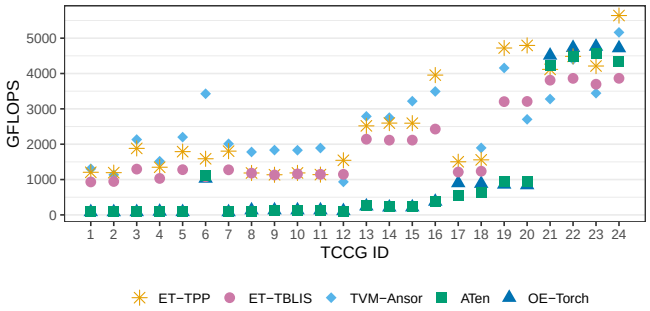


Figure 4. Performance of the five studied approaches for the modified TCCG settings on the SPR testbed. Performance is expressed in FP32 GFLOPS.

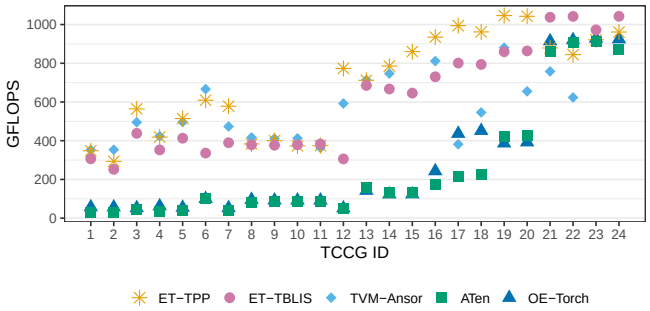


Figure 5. Performance of the five studied approaches for the modified TCCG settings on the Ryzen testbed. Performance is expressed in FP32 GFLOPS.

settings 13-24. This may be due to the increasing tensor contraction sizes for higher TCCG IDs, which effectively increases the size of the search space for Ansor. On the SPR and Ryzen testbeds (Fig. 4 and Fig. 5), the advantages of TVM-Ansor for settings 1-12 are less pronounced. One explanation could be optimization challenges due to the increased vector widths on the two x86 systems: Grace has 128-bit pipelines, while Ryzen has 256-bit vector units, and SPR 512-bit units.

In any case, the achieved performance of TVM-Ansor must also be put in relation to the required tuning time, which is several orders of magnitude higher than the compilation time of all other approaches. Specifically, on the Grace testbed, Ansor required between 1103 s (ID 6) and 10932 s (ID 23) to optimize a single contraction. In contrast, ET-TPP took between 0.4 ms (ID 16) and 3.3 ms (ID 21) to optimize a single contraction. Note that we aborted TVM-Ansor's tuning process for settings 23 and 24 on the Ryzen testbed after 12 hours of runtime.

The two approaches, ATen and OE-Torch, show similar behavior on all three testbeds. For the settings with low arithmetic intensity, their performance is only a fraction of ET-TPP, ET-TBLIS and TVM-Ansor. Only for the computationally intensive settings 21-24 do they reach a performance close to the other approaches. The reason for this observation lies in the strategy of mapping the contractions onto large GEMMs according to the TTGT approach. This requires explicit permutations of the tensors in memory and is amortized only for large contractions with high arithmetic intensity.

ET-TPP has competitive performance on all three testbeds and outperforms the other approaches in many settings. The two TCCG settings 17 and 18 are outliers on the two server systems (Grace and SPR). This observation can be explained by our simple lowering approach to nested loops over small GEMMs. Taking setting 17, i.e., the contraction gkiljh,ekilfj→efgh as an example, we identify the dimensions f, h, and j as GEMM dimensions and the remaining ones as loop dimensions. Of the loop dimensions, only e and g are of type M or N, while the others are of type K. Considering the dimension sizes, we see that our OpenMP parallelization is limited by the $|e| \cdot |g| = 6 \cdot 4 = 24$ primitive executions, which is not enough to utilize all 72 Grace cores or 48 SPR cores. In such a case, one should deviate from our simple lowering scheme and reduce the load per primitive execution to increase the amount of available multicore parallelism.

6.4 Contraction Trees

We now examine the performance of the proposed approach on a set of complex contraction trees.

Tab. 5 summarizes the studied settings by their einsum expression, dimension sizes and contraction path. As for the binary contractions in Sec. 6.3, the dimension sizes are given in alphabetical order, with uppercase IDs appearing before lowercase ones. The contraction path follows the notation introduced in Sec. 2.2.

In addition, we provide a brief high-level description of the settings and refer interested readers to the cited works for a full motivation and detailed description of the settings:

SYN Synthetic contraction tree used for the example optimization shown in Fig. 2a.

TT Tensor train decomposition of the COIL-100 dataset [46] using the tntorch software [54] with a relative error below $\epsilon = 0.2$.

FCTN Fully connected tensor network decomposition of a hyperspectral video with dimensions $60 \times 60 \times 20 \times 20$, where all FCTN ranks are set to 8 [61].

TW Tensor wheel decomposition of a hyperspectral video of dimensions $40 \times 40 \times 20 \times 20$ with TW ranks $R_1 = R_2 = R_3 = R_4 = 6$ and $L_1 = L_2 = L_3 = L_4 = 4$ [56].

GETD Tensor ring part of a generalized model based on Tucker decomposition and tensor ring decomposition applied to the JF17K-4 dataset [34].

TRN First layer of a tensor ring network compressing LeNet-300-100 with all ranks set to 50 and a batch size of 128 [2].

Table 5. Benchmarked contraction trees given by their einsum expression, dimension sizes, and contraction path. All settings were benchmarked using FP32 arithmetic.

ID	Einsum String / Dim. Sizes / Contraction Path
SYN	iaje,bf,dcba,cigj,dh→hgfei 24,48,12,56,32,64,8,84,8,72 (1,2),(2,3),(0,1),(0,1)
TT	af,fbg,gch,hdi,ie→abcde 100,72,128,128,3,71,305,32,3 (1,2),(0,3),(0,1),(0,1)
FCTN	aefg,behi,cfhj,dgij→abcd 60,60,20,20,8,8,8,8,8,8 (2,3),(0,2),(0,1)
TW	aefi,bfgj,cghk,dhel,ijkl→abcd 40,40,20,20,6,6,6,6,4,4,4,4 (2,3),(2,3),(0,2),(0,1)
GETD	aib,bjc,ckd,dle,ema→ijklm 40,40,40,40,40,25,25,25,25,25 (0,1),(0,1),(0,1),(0,1)
TRN	Babcd,aij,bik,ckl,dlm,ejp,fop,gno,hmn→Befgh 128,4,7,4,7,3,4,5,5,50,50,50,50,50,50,50 (1,5),(4,5),(1,2),(1,2),(1,2),(1,2),(1,2),(0,1)
MERA	Setting str_nw_mera_open_26 [5]. 26 input tensors, dimension sizes from 3-68. Intermediate size optimized contraction path.
TNLM	Setting lm_first_last_brackets_4_16d [5]. 52 input tensors, dimension sizes from 7-16. Intermediate size optimized contraction path.

MERA Multi-Scale entanglement renormalization ansatz with open legs, used to model boundary effects and capture correlations and observables in complex systems [5].

TNLM Tensor network language model inference query that computes the joint probability distribution of the first and last tokens in a context [5].

As already done for the binary nodes in Sec. 6.3, we ran ET-TPP, ET-TBLIS, ATen, and OE-Torch 100 times and report the mean performance. For TVM-Ansor, we again report the median of the built-in time evaluator with a minimum repetition time of 60 s. Tab. 6 shows the performance of the studied approaches for the eight contraction trees on the three testbeds. Settings that did not run or did not finish within 12 hours are marked as "N/A". In the case of TVM-Ansor, the TRN tunings did not finish within 12 hours and the two largest tunings (MERA and TNLM) ran out of memory. ATen only supports A-Z and a-z as dimension IDs, which is insufficient for TNLM.

Examining the results, we see that ET-TPP outperforms the other approaches in most cases. These results are especially encouraging considering our simple IR tree optimization

Table 6. Sustained FP32 GFLOPS for the benchmarked contraction trees. The "Speedup" rows show the performance improvement of ET-TPP over the other approaches.

ID	Approach	Grace	SPR	Ryzen
SYN	ET-TPP	**4838**	**4790**	**942**
	ET-TBLIS	1901	2189	781
	TVM-Ansor	3172	3151	678
	ATen	778	814	443
	OE-Torch	632	1152	492
	Speedup	**1.5×**	**1.5×**	**1.2×**
TT	ET-TPP	**3315**	**2591**	**562**
	ET-TBLIS	2455	1634	531
	TVM-Ansor	2018	1291	293
	ATen	1403	262	162
	OE-Torch	485	249	135
	Speedup	**1.4×**	**1.6×**	**1.1×**
FCTN	ET-TPP	**3092**	**2295**	**667**
	ET-TBLIS	1322	2109	527
	TVM-Ansor	2578	1582	459
	ATen	409	238	90
	OE-Torch	346	250	169
	Speedup	**1.2×**	**1.1×**	**1.3×**
TW	ET-TPP	**1878**	746	**847**
	ET-TBLIS	173	524	542
	TVM-Ansor	1195	**1035**	525
	ATen	114	43	91
	OE-Torch	57	130	111
	Speedup	**1.6×**	0.7×	**1.6×**
GETD	ET-TPP	**4747**	**3677**	700
	ET-TBLIS	4343	2936	**839**
	TVM-Ansor	2719	2497	637
	ATen	1647	707	366
	OE-Torch	826	930	445
	Speedup	**1.1×**	**1.3×**	0.8×
TRN	ET-TPP	**1963**	**1057**	374
	ET-TBLIS	933	810	**770**
	TVM-Ansor	N/A	N/A	N/A
	ATen	243	227	405
	OE-Torch	51	361	461
	Speedup	**2.1×**	**1.3×**	0.5×
MERA	ET-TPP	**4025**	**3238**	**815**
	ET-TBLIS	1958	2052	796
	TVM-Ansor	N/A	N/A	N/A
	ATen	1346	452	298
	OE-Torch	703	594	298
	Speedup	**2.1×**	**1.6×**	**1.0×**
TNLM	ET-TPP	**4378**	3095	900
	ET-TBLIS	3159	**3608**	**991**
	TVM-Ansor	N/A	N/A	N/A
	ATen	N/A	N/A	N/A
	OE-Torch	3103	3532	746
	Speedup	**1.4×**	0.9×	0.9×

heuristic and straightforward lowering to nested loops over small GEMMs.

ATen and OE-Torch, two popular options for computing einsum expressions, are significantly slower than ET-TPP and ET-TBLIS. TVM-Ansor delivers only mediocre performance for the settings studied, possibly due to the further increased search space size compared to the single contractions in Sec. 6.3. Only one configuration (TW on SPR) has TVM-Ansor in the lead.

ET-TBLIS is able to outperform ET-TPP for the GETD, TRN and TNLM settings on the Ryzen testbed. This observation can be explained by examining the optimized Einsum Tree IR and the resulting mapping of binary nodes to loops over tensor processing primitives. Using the TRN setting as an example, the compiled IR tree performs 53% of all operations in the following binary node: cbdinh,eaifgn→eacbdfgh. We see that the corresponding GEMM primitive becomes $C[fg][h]$ += $A[n][h] * B[fg][n]$ with $|h| = 5$, $|f| \cdot |g| = 20$, $|n| = 50$. The performance-limiting property is the small size of the h dimension that the code generators use for vectorization. As we saw in Sec. 6.2, a good size for fast kernels would be $|h| = 32$. Possible future extensions of the ET-TPP approach could address such cases on several levels: First, one could develop more sophisticated heuristics that prioritize computationally expensive contractions in Einsum Tree IR. Second, we could insert interior unary nodes to perform permutations in such cases. The unary nodes could, depending on their complexity, also be efficiently fused into the preceding or following binary nodes.

Table 7. Compilation time for ET-TPP, ET-TBLIS, TVM-Ansor, and OE-Torch. All measurements were taken on the Grace system.

ID	ET-TPP	ET-TBLIS	TVM-Ansor	OE-Torch
SYN	0.78 ms	0.54 ms	37.83 m	0.17 ms
TT	2.16 ms	0.55 ms	66.92 m	0.17 ms
FCTN	0.64 ms	0.41 ms	25.52 m	0.15 ms
TW	0.85 ms	0.47 ms	42.17 m	0.18 ms
GETD	0.85 ms	0.39 ms	39.68 m	0.17 ms
TRN	0.78 ms	0.50 ms	N/A	0.26 ms
MERA	3.84 ms	1.26 ms	N/A	0.64 ms
TNLM	8.79 ms	1.26 ms	N/A	1.37 ms

Finally, we report the compilation times for the studied approaches on the Grace system in Tab. 7. ATen is excluded because at::einsum parses the function parameters and executes the contraction tree in a single step. We see that OE-Torch is the fastest approach, followed by ET-TBLIS and ET-TPP. The difference between ET-TPP and ET-TBLIS can be explained by the just-in-time kernel generation of ET-TPP by the LIBXSMM library. In contrast, ET-TBLIS relies on precompiled microkernels. In general, ET-TPP offloads more specialization to the compilation phase while ET-TBLIS and OE-Torch make some of these decisions at runtime in the TBLIS and PyTorch backends. TVM-Ansor takes minutes to tune the studied trees, while the other approaches are in the millisecond range.

7 Related Work

This section is structured into three parts: the first part covers research on two-input tensor contractions, the second part addresses contractions with more than two input tensors, and the third part explores methods for decomposing tensor contractions into primitives.

Single-contraction approaches. For contractions involving two low-order tensors, such as vectors or matrices, tuned BLAS libraries [17, 55] are generally the optimal choice. However, when dealing with higher-order tensors, BLAS libraries alone are insufficient. To address this shortcoming, previous research on tensor contractions has established four primary approaches. (1) nested-loop code [4, 35, 38, 51] relies on basic loops but suffers from inefficiencies with large tensors and strided memory access. These inefficiencies can be partly mitigated by vectorization [20] or proper loop tiling [33]. (2) Transpose-Transpose-GEMM-Transpose (TTGT) [11, 26] leverages GEMM efficiency but incurs transposition overhead. (3) Loops-over-GEMMs (LoG) [9, 30, 47] breaks down contractions into smaller GEMM calls but may struggle with small or strided submatrices. (4) GEMM-like Tensor-Tensor Multiplication (GETT) [49] achieves high efficiency by using packed sub-tensors aligned with the CPU cache hierarchy. GETT acts as a fused TTGT approach, integrating transpositions directly into sub-tensor packing, thus avoiding extra memory usage. Similarly, TBLIS [37] improves memory efficiency and speed through dynamic tensor layout transformations.

Our approach cannot be directly classified into any of these categories, as it adopts a global perspective on tensor expressions beyond the two-tensor case (see Fig. 6). However, it integrates concepts from TTGT and LoG. By inserting permutation nodes (TTGT), we bring input tensors into a form that enables the efficient use of small GEMMs (LoG) for subsequent tensor contractions. Note, however, unlike TTGT, our approach requires permutations only for the initial input tensors, not intermediate tensor contractions, and avoids LoG's common limitations like unfavorable strided submatrices.

Multiple-contraction approaches. For tensor expressions with more than two tensors, computations are typically reduced to binary contractions along an optimized contraction path, as contracting all tensors at once would be highly inefficient. These binary contractions are commonly executed using one of the *single-contraction approaches* discussed above. For example, ITensor [13], a C++ and Julia

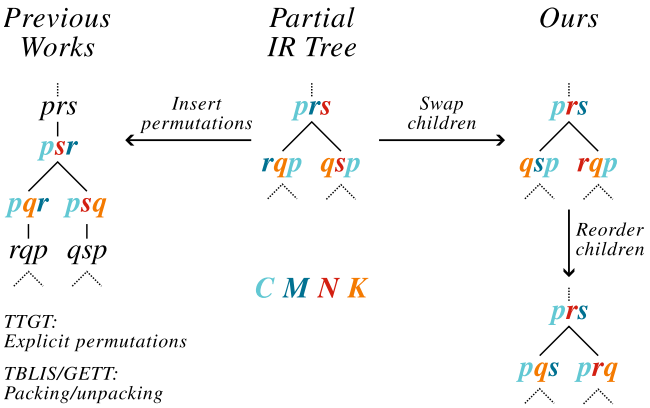


Figure 6. Shown is a comparison between previous works on binary tensor contractions and our tree-based approach. The binary approaches have a limited view of the problem and require tensor permutations (transpositions), either performed explicitly or through packing/unpacking routines. Our approach offloads the optimization of the data layout to the preceding contractions. The colors illustrate the dimension types introduced in Sec. 2.1.

tensor network library, defaults to the TTGT approach for binary contractions and offers TBLIS as an experimental backend. However, global strategies beyond binary contractions are not employed in ITensor. Similarly, the Cyclops Tensor Framework [48], a C++ and Python library for multidimensional arrays, computes binary contractions using the TTGT approach. Also, TTGT is the preferred approach for multiple binary contractions in PyTorch [42], specifically in its ATen tensor library [44]. However, instead of GEMM, PyTorch uses batch matrix multiplication to support indices shared by both input tensors and the output tensor.

To our knowledge, aside from common subexpression identification [21], there are no multiple-contraction approaches that adopt a global perspective on optimizing both the memory layout of tensors and computations across entire tensor expressions for efficient binary contractions along a contraction path. However, some deep learning frameworks, while not strictly focusing on contractions, employ global strategies to execute multiple tensor operations with minimal overhead. We consider these frameworks and their strategies related to our approach and thus mention them here. SmartMem [39] aims to eliminate layout transformations between the computational operators in transformers by both adapting the tensor layout and the operators. Korch [23] optimizes operator fusion within deep neural networks (DNNs) by first applying operator fission to decompose tensor operators into a compact set of basic tensor algebra primitives, thus enabling a range of fine-grained, inter-operator optimizations. Souffle [57], an open-source compiler, accelerates DNN inference by building a global tensor dependency graph that enables cross-operator optimizations, segmenting the computation graph into subprograms, and applying local optimizations to enhance instruction-level parallelism and data reuse. Finally, oneDNN Graph [31] introduces a hybrid tensor compiler that combines compiler optimizations with expert-tuned kernels to achieve high-performance code generation for DNN computation graphs, addressing specific deep learning challenges like low-precision computation, aggressive graph operation fusion, static tensor shape optimization, and memory reuse.

Primitive-based approaches. Approaches based on primitives decompose tensor computations into a set of tensor processing primitives. These primitives can be either handcrafted by humans or generated, that is lowered, through human-engineered code generators. Primitive collections are available for various hardware architectures, including CUTLASS [25] and Thunderkittens [3] for NVIDIA GPUs, CK [6] for AMD GPUs, KleidiAI [24] for Arm CPUs, and XNNPACK [59], which additionally supports RISC-V CPUs and WebAssembly environments. Having a set of ready-to-use primitives frees developers from writing architecture-specific instruction-level optimizations and enables them to compose operations beyond those offered by vendor-provided tensor libraries. For example, PyTorch uses Triton [53], a compiler for creating efficient deep learning primitives on GPUs, enabling dynamic generation of optimized GPU kernels for tasks like matrix multiplications and convolutions without low-level CUDA programming. PyTorch also supports lowering to KleidiAI and XNNPACK. ParLooper [15], a framework for generating complex nested loop computations, decomposes tensor operations on CPUs into LIBXSMM tensor processing primitives. In our approach, einsum trees are optimized and then lowered into a series of nested loops over primitives. Similar to ParLooper, we use LIBXSMM to lower these primitives to hardware. However, because einsum trees serve as a high-level intermediate representation, our approach can be adapted to use other primitive collections, such as those mentioned above, and potentially even lower through Triton or ParLooper, thereby extending its applicability beyond CPUs and the optimizations provided by LIBXSMM. In the long term, we also see the MLIR ecosystem as a suitable host for this work, as it provides a rich set of tools for lowering high-level intermediate representations to hardware primitives [16].

8 Discussion and Outlook

We have introduced the high-level intermediate representation Einsum Tree IR, which is heavily inspired by the formalism of contraction trees that appear in the context of einsum evaluations. Our optimization heuristic for trees in the IR is remarkably simple, deriving a formulation that can be lowered to nested loops over two tensor processing primitives: Small matrix transpositions and small (packed) GEMMs. The low-level implementation of the primitives is

straightforward, requiring only appropriate register blocking. Advanced techniques commonly used in BLAS libraries, such as cache blocking or packing/unpacking routines, are not required for their implementation. In conclusion, the presented approach drastically reduces the effort required to port the efficient computation of complex tensor contraction trees to new instruction set extensions.

We demonstrated the effectiveness of our approach by evaluating it on three different hardware testbeds: NVIDIA Grace CPU, Intel Xeon Platinum 8488C, and AMD Ryzen 7 8700G. Our approach is able to achieve very high floating point throughput on the three systems. On a set of real-world and complex contraction trees, we outperformed four state-of-the-art software packages in most cases with sustained speedups of up to 2.1×.

We see Einsum Tree IR as the backbone for a rich set of future developments. Several directions for future work are possible for optimizations in Einsum Tree IR and in lowering the nodes in the tree to nested loops over tensor processing primitives. Tree-level optimizations can prioritize computationally expensive contractions and target a more local search for good layouts of intermediate steps. There are also degrees of freedom in the lowering and scheduling step, where, for example, future extensions could balance trade-offs between identifying high-performance primitives and available multicore parallelism.

Another natural next step of this work would be to integrate Einsum Tree IR with the emerging compiler ecosystems, such as MLIR. A future Einsum Tree MLIR dialect in conjunction with a TPP dialect may be a viable avenue for such an endeavor. At the same time, this could formalize the approach of having different backends, e.g., by providing a lowering option to LinAlg. In addition, we foresee novel matrix accelerators being addressed by new masks for the (packed) GEMM primitive. Such optimizations could target Intel AMX or Arm matrix instructions with built-in dot products.

Acknowledgements

This work was supported by Carl-Zeiss-Stiftung as part of the "Interactive Inference" project.

A Artifact Appendix

A.1 Abstract

The software einsum_ir was used to obtain the results ET-TPP, ET-TBLIS and ATen in the paper. The source code of einsum_ir is available at https://zenodo.org/records/14843572. We describe the steps necessary to obtain raw performance results for ET-TPP and ATen. In addition, we briefly describe the post-processing pipeline used to obtain the numbers reported in Tab. 3, Fig. 3, Fig. 4, Fig. 5, Tab. 6.

We share scripts, raw results, post-processed results, and our documentation of the runs in a separate data repository. This includes the setup of the TBLIS backend in einsum_ir (ET-TBLIS) as well as tools and configurations to run TVM AutoScheduler (TVM-Ansor) and opt_einsum with PyTorch as backend (OE-Torch). The three directories `grace`, `spr` and `ryzen_8700g` of the data repository contain the setups and raw results for the hardware testbeds: Grace, SPR, and Ryzen in the paper. Note that some of the studies in the data repository are not included in the paper. For example, einsum_ir also supports BLAS libraries as a backend for the GEMM primitive. These studies are included in the repository, but omitted from the paper due to space limitations. The data repository is available at https://zenodo.org/records/14843576.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Optimization and lowering of Einsum Tree IR.
- **Program:** einsum_ir.
- **Compilation:** C/C++ compiler.
- **Run-time environment:** Linux operating system.
- **Hardware:** NVIDIA Grace CPU, Intel Xeon Platinum 8488C, AMD Ryzen 7 8700G SoC.
- **Metrics:** FP32 GFLOPS for evaluating a compiled einsum tree. Time for compilation itself.
- **Output:** Raw logging output from the software.
- **Experiments:** 1) Binary tensor contractions (Tab. 4). 2) Contraction trees (Tab. 5).
- **How much disk space required (approximately)?:** Less than 50 GiB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** Less than two days per hardware configuration including 100 repetitions and three einsum_ir backends.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Workflow automation framework used?:** Shell and Python scripts (semi-automated).

A.3 Description

A.3.1 How to access.

- Git repository: https://github.com/scalable-analyses/einsum_ir
- Git hash of benchmarked commit: c600e8a.

A.3.2 Hardware dependencies.

- We used the following hardware for the experiments in the paper: NVIDIA Grace CPU (Grace), Intel Xeon Platinum 8488C (SPR), AMD Ryzen 7 8700G SoC (Ryzen). The tested machines have at least 64 GiB of memory.
- The main software contribution, einsum_ir with the TPP backend (ET-TPP), should run on most CPUs due to support for many vector extensions in the primitive-providing LIBXSMM dependency.

- Most settings will run on machines with less than 64 GiB of memory. The TNLM contraction tree has the largest memory footprint.

A.3.3 Software dependencies.

- LIBXSMM (required): https://github.com/libxsmm/libxsmm.
- PyTorch (required for `bench_expression`): https://pytorch.org.
- Catch2 (optional): https://github.com/catchorg/Catch2

A.4 Installation

The basic installation procedure on a Linux-based system is as follows:

```
# install scons and torch
python -m venv venv_pytorch
source venv_pytorch/bin/activate
pip install scons
pip install torch==2.5.1 torchvision torchaudio \
--index-url https://download.pytorch.org/whl/cpu

# install libxsmm
git clone https://github.com/libxsmm/libxsmm.git
cd libxsmm; git checkout 6a286ec
make BLAS=0 -j
cd ..

# install einsum_ir
git clone https://github.com/scalable-analyses/einsum_ir
cd einsum_ir; git checkout c600e8a

CATCH2_REPO="https://github.com/catchorg/Catch2"
wget ${CATCH2_REPO}/releases/download/v2.13.10/catch.hpp
LIBTORCH=$(python -c "import sysconfig;"\
"print(sysconfig.get_paths()['purelib'])")
ln -s ${LIBTORCH}/torch.libs/* ${LIBTORCH}/torch/lib

CXX=clang++ CC=clang scons libtorch=${LIBTORCH}/torch \
libxsmm=$(pwd)/../libxsmm -j8
```

The installation scripts we used for the experiments in the paper can be found in the data repository. For example, the script `spr/install.sh` contains all the commands to install einsum_ir on the SPR system starting with a standard Fedora Cloud 41 AMI.

A.5 Experiment workflow

The configurations used for the experiments in the paper are located in the `samples` directory of the einsum_ir repository. For example, we can run the FCTN setting in `samples/tensor_decomp/fctn.cfg` by simply passing the config in that file to `bench_expression`:

```
./build/bench_expression "aefg,behi,cfhj,dgij->abcd" \
"60,60,20,20,8,8,8,8,8,8" "(2,3),(0,2),(0,1)"
```

The paper reports ET-TPP results using one thread per core for Grace and Ryzen, and two threads per core for the SPR system. This is 72 OMP threads for the Grace system, 96 threads for the SPR system and eight threads for the Ryzen system. We pinned the threads using the `KMP_AFFINITY` environment variable. For example, on the Ryzen system, we ran `bench_expression` as follows:

```
KMP_AFFINITY="explicit,proclist=[0-7],granularity=fine"\
 OMP_NUM_THREADS=8 ./build/bench_expression #[...]
```

The behavior of einsum_ir can be further influenced by using two environment variables. `EINSUM_IR_BACKEND` can be used to switch between the TPP, BLAS and TBLIS backends of the software. `EINSUM_IR_REORDER_DIMS` can be used to disable the optimization heuristic described in Sec. 5 and corresponding thresholds in Sec. 6.2. For the experiments in the paper, we used the convenience script `samples/tools/bench_einsum_ir.sh`. The script automates the benchmarking process by setting the necessary environment variables and repeatedly running different settings:

```
bash einsum_ir/samples/tools/bench_einsum_ir.sh \
-e einsum_ir/build/bench_expression \
-l logs -r 100 -b tpp -d 1 \
-k syn,tt,fctn,tw,getd,trn,mera,tnlm,tccg_blocked
```

A comprehensive list of the tests we ran is part of the data repository. For example, the script `spr/run_einsum_ir.sh` contains all the runs performed on the SPR system.

A.6 Evaluation and expected results

The `bench_expression` program reports basic information about the parsed and optimized einsum_tree, as well as the benchmarked performance and some verification results. For the described execution of the FCTN setting on Grace, the output will look similar to the following:

```
parsed tensors:
  a,e,f,g
  b,e,h,i
  c,f,h,j
  d,g,i,j
  a,b,c,d
parsed contraction path: 2 3 0 2 0 1
...
parsed dimension sizes:
  a: 60
  b: 60
  c: 20
...
*** benchmarking einsum_ir ***
...
  #flops:         3058272000
  time (compile): 0.000597352
  time (eval):    0.00100908
  gflops (eval):  3030.76
  gflops (total): 1903.77
```

```
CSV_DATA: einsum_ir,"aefg,behi,cfhj,dgij->abcd
     ","60,60,20,20,8,8,8,8,8,8","(2,3),(0,2),(0,1)
     ",3058272000,0.000597352,0.00100908,3030.76,1903.77

*** benchmarking at::einsum ***
...
CSV_DATA: at::einsum,"aefg,behi,cfhj,dgij->abcd
     ","60,60,20,20,8,8,8,8,8,8","(2,3),(0,2),(0,1)
     ",3058272000,0,0,0,349.362

*** benchmarking at::matmul ***
  C M N K for the binary contractions:
    #0: 1 1280 1280 8
    #1: 1 25600 480 64
    #2: 1 24000 60 512
    #flops:         3058272000
    time (eval):    0.00321821
    gflops (eval):  950.304
CSV_DATA: at::matmul,"aefg,behi,cfhj,dgij->abcd
     ","60,60,20,20,8,8,8,8,8,8","(2,3),(0,2),(0,1)
     ",3058272000,0,0.00321821,950.304,0
...
```

We see that einsum_ir (ET-TPP in the paper) achieved a evaluation performance of 3031 FP32 GFLOPS and that at::einsum (ATen in the paper) reached 349 GFLOPS. The results for at::matmul are not reported in the paper. Here we assume a Transpose-Transpose-GEMM-Transpose (TTGT) approach, but only benchmark the time of the standalone BGEMMs using at::matmul. This effectively assumes zero overhead permutations in the TTGT approach (see Fig. 6).

We used the `CSV_DATA` lines to post-process the raw results. The post-processing scripts are located in the `analysis` directory of the data repository. We used the `summarize_all.sh` script to collect the `CSV_DATA` lines from the raw log files and get statistics over the repeated runs. The other scripts in the `analysis` directory generate plots and tables.

A.7 Experiment customization

The experiments can be customized in several ways:

- The derivation of the contraction trees in Tab. 5 is documented in the `samples` directory of einsum_ir. New trees can be derived in a similar way.
- ET-TPP relies on LIBXSMM to generate fast kernels. Many CPU architectures that are not part of our experiments should work out of the box.
- The `bench_expression` program has some command line options, such as using FP64 arithmetic instead of FP32. Try `bench_expression --help`.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Vaneet Aggarwal, Wenlin Wang, Brian Eriksson, Yifan Sun, and Wenqi Wang. 2018. Wide Compression: Tensor Ring Nets. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9329–9338. https://doi.org/10.1109/CVPR.2018.00972

[3] Anonymous. 2024. ThunderKittens: Simple, Fast, and $\textit{Adorable}$ Kernels. In *Submitted to The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=0fJfVOSUra under review.

[4] Edoardo Aprà, Michael Klemm, and Karol Kowalski. 2014. Efficient Implementation of Many-Body Quantum Chemical Methods on the Intel® Xeon Phi Coprocessor. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 674–684. https://doi.org/10.1109/SC.2014.60

[5] Mark Blacher, Christoph Staudt, Julien Klaus, Maurice Wenig, Niklas Merk, Alexander Breuer, Max Engel, Sören Laue, and Joachim Giesen. 2024. Einsum Benchmark: Enabling the Development of Next-Generation Tensor Execution Engines. In *Proceedings of the Annual Conference on Neural Information Processing Systems, Track on Datasets and Benchmarks (NeurIPS)*.

[6] Liu Chao. 2022. *AMD Composable Kernel library: efficient fused kernels for AI apps with just a few lines of code.* Technical Report. Advanced Micro Devices Inc.

[7] NVIDIA Corporation. 2024. cuQuantum SDK: A High-Performance Library for Accelerating Quantum Science. https://docs.nvidia.com/cuda/cuquantum/latest/index.html.

[8] G Daniel, Johnnie Gray, et al. 2018. Opt_einsum - a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software* 3, 26 (2018), 753.

[9] Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Ortí, and Paolo Bientinesi. 2014. Towards an efficient use of the BLAS library for multilinear tensor contractions. *Appl. Math. Comput.* 235 (2014), 454–468. https://doi.org/10.1016/j.amc.2014.02.051

[10] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.

[11] Evgeny Epifanovsky, Michael Wormit, Tomasz Kus, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. 2013. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *J. Comput. Chem.* 34, 26 (2013), 2293–2309.

[12] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 804–817.

[13] Matthew Fishman, Steven White, and Edwin Stoudenmire. 2022. The ITensor software library for tensor network calculations. *SciPost Physics Codebases* (2022), 004.

[14] Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. 2020. Harnessing Deep Learning via a Single Building Block. In *2020 IEEE International Parallel and Distributed*

Processing Symposium (IPDPS). 222–233. https://doi.org/10.1109/IPDPS47924.2020.00032

[15] Evangelos Georganas, Dhiraj Kalamkar, Kirill Voronin, Abhisek Kundu, Antonio Noack, Hans Pabst, Alexander Breuer, and Alexander Heinecke. 2024. Harnessing Deep Learning and HPC Kernels via High-Level Loop and Tensor Abstractions on CPU Architectures. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 950–963.

[16] Renato Golin, Lorenzo Chelini, Adam Siemieniuk, Kavitha Madhu, Niranjan Hasabnis, Hans Pabst, Evangelos Georganas, and Alexander Heinecke. 2024. Towards a high-performance AI compiler with upstream MLIR. *arXiv preprint arXiv:2404.15204* (2024).

[17] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 1–25.

[18] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. *Quantum* 5 (2021), 410. https://doi.org/10.22331/q-2021-03-15-410

[19] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, et al. 2020. Array programming with NumPy. *Nature* (2020).

[20] Robert J. Harrison, Gregory Beylkin, Florian A. Bischoff, Justus A. Calvin, George I. Fann, Jacob Fosso-Tande, Diego Galindo, Jeff R. Hammond, Rebecca Hartman-Baker, Judith C. Hill, Jun Jia, Jakob S. Kottmann, Miao-Jung Yvonne Ou, Junchen Pei, Laura E. Ratcliff, Matthew G. Reuter, Adam C. Richie-Halford, Nichols A. Romero, Hideo Sekino, William A. Shelton, Bryan E. Sundahl, W. Scott Thornton, Edward F. Valeev, Álvaro Vázquez-Mayagoitia, Nicholas Vence, Takeshi Yanai, and Yukina Yokoi. 2016. MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation. *SIAM J. Sci. Comput.* 38, 5 (2016).

[21] Albert Hartono, Qingda Lu, Thomas Henretty, Sriram Krishnamoorthy, Huaijian Zhang, Gerald Baumgartner, David E. Bernholdt, Marcel Nooijen, Russell Pitzer, J. Ramanujam, and P. Sadayappan. 2009. Performance Optimization of Tensor Contraction Expressions for Many-Body Methods in Quantum Chemistry. *The Journal of Physical Chemistry A* 113, 45 (2009), 12715–12723.

[22] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[23] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, et al. 2024. Optimal Kernel Orchestration for Tensor Programs with Korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 755–769.

[24] Gian Marco Iodice. 2024. *Arm KleidiAI: Helping AI frameworks elevate their performance on Arm CPUs*. Technical Report. Arm.

[25] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. 2017. *Cutlass: Fast linear algebra in cuda c++*. Technical Report. NVIDIA.

[26] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500.

[27] Jean Kossaifi, Zachary C. Lipton, Arinbjörn Kolbeinsson, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. 2020. Tensor Regression Networks. *Journal of Machine Learning Research* 21 (2020), 123:1–123:21.

[28] Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. 1997. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Parallel Process. Lett.* (1997).

[29] Sören Laue, Matthias Mitterreiter, and Joachim Giesen. 2020. A Simple and Efficient Tensor Calculus. In *AAAI Conference on Artificial Intelligence (AAAI)*. 4527–4534.

[30] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard W. Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 76:1–76:12.

[31] Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, et al. 2024. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 460–470.

[32] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[33] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P. Sadayappan. 2019. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, 74:1–74:13.

[34] Yu Liu, Quanming Yao, and Yong Li. 2020. Generalizing Tensor Decomposition for N-ary Relational Knowledge Bases. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 1104–1114. https://doi.org/10.1145/3366423.3380188

[35] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, and Karol Kowalski. 2011. GPU-based implementations of the noniterative regularized-CCSD (T) corrections: applications to strongly correlated systems. *Journal of chemical theory and computation* 7, 5 (2011), 1316–1327.

[36] Igor L. Markov and Yaoyun Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (2008), 963–981.

[37] Devin A Matthews. 2018. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24.

[38] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary W. Hall, Paul D. Hovland, Elizabeth R. Jessup, and Boyana Norris. 2015. Generating Efficient Tensor Contractions for GPUs. In *International Conference on Parallel Processing, ICPP*. IEEE Computer Society, 969–978. https://doi.org/10.1109/ICPP.2015.106

[39] Wei Niu, Md Musfiqur Rahman Sanim, Zhihao Shu, Jiexiong Guan, Xipeng Shen, Miao Yin, Gagan Agrawal, and Bin Ren. 2024. SmartMem: Layout Transformation Elimination and Adaptation for Efficient DNN Execution on Mobile. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 916–931.

[40] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. 2015. Tensorizing Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 442–450.

[41] Román Orús. 2019. Tensor networks for complex quantum systems. *Nature Reviews Physics* (2019).

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Neural Information Processing Systems (NeurIPS)*.

[43] Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. 2020. Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits. In *Proceedings of the International Conference on Machine Learning (ICML)*. 7563–7574.

[44] PyTorch Developers. 2024. ATen, PyTorch's tensor library. https://github.com/pytorch/pytorch/tree/main/aten/src/ATen.

[45] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*. 126–132.

[46] S. K. Nayar S. A. Nene and H. Murase. 1996. *Columbia Object Image Library (COIL-100)*. Technical Report. Columbia University.

[47] Yang Shi, U. N. Niranjan, Animashree Anandkumar, and Cris Cecka. 2016. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. (2016), 193–202.

[48] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.

[49] Paul Springer and Paolo Bientinesi. 2018. Design of a high-performance GEMM-like tensor–tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)* 44, 3 (2018), 1–29.

[50] Christoph Staudt, Mark Blacher, Julien Klaus, Farin Lippmann, and Joachim Giesen. 2024. Improved Cut Strategy for Tensor Network Contraction Orders. In *Proceedings of the International Symposium on Experimental Algorithms (SEA)*. 27:1–27:19.

[51] Kevin Stock, Tom Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert Harrison. 2011. Model-Driven SIMD Code Generation for a Multi-resolution Tensor Kernel. In *2011 IEEE International Parallel and Distributed Processing Symposium*. 1058–1067. https://doi.org/10.1109/IPDPS.2011.101

[52] Edwin Miles Stoudenmire and David J. Schwab. 2016. Supervised Learning with Tensor Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 4799–4807.

[53] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[54] Mikhail Usvyatsov, Rafael Ballester-Ripoll, and Konrad Schindler. 2022. tntorch: Tensor Network Learning with PyTorch. *Journal of Machine Learning Research* 23, 208 (2022), 1–6. http://jmlr.org/papers/v23/21-1197.html

[55] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–33.

[56] Zhong-Cheng Wu, Ting-Zhu Huang, Liang-Jian Deng, Hong-Xia Dou, and Deyu Meng. 2022. Tensor wheel decomposition and its tensor completion application. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1958, 13 pages.

[57] Chunwei Xia, Jiacheng Zhao, Qianqi Sun, Zheng Wang, Yuan Wen, Teng Yu, Xiaobing Feng, and Huimin Cui. 2024. Optimizing deep learning inference via global analysis and tensor expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 286–301.

[58] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 684–691. https://doi.org/10.1109/ICPADS.2012.97

[59] XNNPACK Developers. 2024. XNNPACK: A Highly Optimized Solution for Neural Network Inference on ARM, x86, WebAssembly, and RISC-V Platforms. https://github.com/google/XNNPACK.

[60] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[61] Yu-Bang Zheng, Ting-Zhu Huang, Xi-Le Zhao, Qibin Zhao, and Tai-Xiang Jiang. 2021. Fully-Connected Tensor Network Decomposition and Its Application to Higher-Order Tensor Completion. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 12 (May 2021), 11071–11078. https://doi.org/10.1609/aaai.v35i12.17321